# Cheat Sheet

## Contents

# Homework

## 1: Speed up on multicore Processors

### Assumptions

- 90% of a given program can be perfectly parrallelized, the other 10% remains sequential
- We have a fixed power budget
- Total power is proportional to the square of frequency
- Performance is proportional to frequency
- The programs performance on a single-core chip is 1 solutions per seconds

### Task 1: Calculate performance on a dual-core chip

On a single-core system, with the frequency $f_1$, has the power usage of $P = cf_1^2$. Since the total power is fixed, the two dual-chip cores must share this power between them. We can therefore calculate the frequency $f_2$ for each of the cores:

$$\frac{P}{2} = cf_2^2$$

$$\frac{cf_1^2}{2} = cf_2^2$$

$$\frac{f_1}{\sqrt{2}} = f_2$$

Assuming that on the sequential sections of the program, only one core is working, we can therefore calculate the performance for the throughput for the parallelizable and sequential sections of the program

$$t_p = 2 * f_2 = 2 * \frac{f_1}{\sqrt{2}} = \sqrt{2} * f_1 = \sqrt{2}\, solutions/s$$

$$t_s = 1 * f_2 = \frac{f_1}{\sqrt{2}} = \frac{1}{\sqrt{2}}\, solutions/s$$

We can now calculate the total troughput, and therefore performance, of the program on the dual-core chip

$$t_{total} = 0.9 * t_p + 0.1 * t_s$$

$$t_{total} = 0.9 * \sqrt{2}\, solutions/s + 0.1 * \frac{1}{\sqrt{2}} solutions/s$$

$$t_{total} = \frac{19\sqrt{2}}{20}\, solutions/s$$

$$t_{total} \approx 1{,}344\, solutions/s$$

## Task 2: Calculate performance on a quad-core chip

As before, we can calculate the frequency $f_4$ for each of the four cores:

$$\frac{P}{4} = cf_4^2$$

$$\frac{cf_1^2}{4} = cf_4^2$$

$$\frac{f_1}{\sqrt{4}} = f_4$$

$$\frac{f_1}{2} = f_4$$

Again, we calculate the performance for the throughput for the parallelizable and sequential sections of the program

$$t_p = 4 * f_4 = 4 * \frac{f_1}{2} = 2f_1 = 2\ solutions/s$$

$$t_s = 1 * f_4 = \frac{f_1}{2} = \frac{1}{2}\ solutions/s$$

We now calculate the total throughput

$$t_{total} = 0.9 * t_p + 0.1 * t_s$$

$$t_{total} = 0.9 * 2\ solutions/s + 0.1 * \frac{1}{2} solutions/s$$

$$t_{total} = 1{,}85\ solutions/s$$

## TA Response

Wrong math on speed averaging. Overall performance does not equal to the sum of the performance times the proportion of each part.

# 2: OpenMP Programming

```
#include <getopt.h>
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
#include <string>

#include <iostream>
#include <sstream>
#include <vector>

#include "./common/CycleTimer.h"
#include "./common/grade.h"
#include "./common/graph.h"
#include "page_rank.h"

#define USE_BINARY_GRAPH 1
#define PageRankDampening 0.3f
#define PageRankConvergence 1e-7d

// used for check correctness
void reference_serial_pageRank(Graph g, double *solution, double damping,
                               double convergence) {
    int numNodes = num_nodes(g);
    double equal_prob = 1.0 / numNodes;
    double *solution_new = new double[numNodes];
    double *score_old = solution;
    double *score_new = solution_new;
    bool converged = false;
    double broadcastScore = 0.0;
    double globalDiff = 0.0;
    int iter = 0;

    for (int i = 0; i < numNodes; ++i) {
        solution[i] = equal_prob;
    }
    while (!converged && iter < MAXITER) {
        iter++;
        broadcastScore = 0.0;
        globalDiff = 0.0;
        for (int i = 0; i < numNodes; ++i) {
            score_new[i] = 0.0;

            if (outgoing_size(g, i) == 0) {
                broadcastScore += score_old[i];
            }
            const Vertex *in_begin = incoming_begin(g, i);
            const Vertex *in_end = incoming_end(g, i);
            for (const Vertex *v = in_begin; v < in_end; ++v) {
                score_new[i] += score_old[*v] / outgoing_size(g, *v);
            }
            score_new[i] =
                damping * score_new[i] + (1.0 - damping) * equal_prob;
        }
        for (int i = 0; i < numNodes; ++i) {
            score_new[i] += damping * broadcastScore * equal_prob;
            globalDiff += std::abs(score_new[i] - score_old[i]);
        }
        converged = (globalDiff < convergence);
        std::swap(score_new, score_old);
    }
    if (score_new != solution) {
        memcpy(solution, score_new, sizeof(double) * numNodes);
    }
    delete[] solution_new;
}
```

```
int main(int argc, char **argv) {

    int num_threads = -1;
    std::string graph_filename;

    if (argc < 3) {
        std::cerr << "Usage: <path/to/graph/file> <manual_set_thread_count>\n";
        exit(1);
    }

    int thread_count = -1;
    if (argc == 3) {
        thread_count = atoi(argv[2]);
    }
    if (thread_count <= 0) {
        std::cerr << "<manual_set_thread_count> must > 0\n";
        exit(1);
    }
    graph_filename = argv[1];

    Graph g;

    printf("----------------------------------------------------------\n");
    printf("Running with %d threads\n", thread_count);
    printf("----------------------------------------------------------\n");
    printf("Loading graph...\n");

    if (USE_BINARY_GRAPH) {
        g = load_graph_binary(graph_filename.c_str());
    } else {
        g = load_graph(argv[1]);
        printf("storing binary form of graph!\n");
        store_graph_binary(graph_filename.append(".bin").c_str(), g);
        delete g;
        exit(1);
    }
    printf("\n");
    printf("Graph stats:\n");
    printf("  Filename: %s\n", argv[1]);
    printf("  Edges: %d\n", g->num_edges);
    printf("  Nodes: %d\n", g->num_nodes);

    bool pr_check = true;
    double *sol1;
    sol1 = (double *)malloc(sizeof(double) * g->num_nodes);
    double *sol2;
    sol2 = (double *)malloc(sizeof(double) * g->num_nodes);

    double pagerank_base;
    double pagerank_time;

    double ref_pagerank_base;
    double ref_pagerank_time;

    double start;
    std::stringstream timing;
    std::stringstream ref_timing;

    timing << "Threads  Page Rank\n";
    ref_timing << "Serial Reference Page Rank\n";

    // Set thread count
    omp_set_num_threads(thread_count);

    // Run implementations
    start = CycleTimer::currentSeconds();
    pageRank(g, sol1, PageRankDampening, PageRankConvergence);
    pagerank_time = CycleTimer::currentSeconds() - start;
```

```
    // Run reference implementation
    start = CycleTimer::currentSeconds();
    reference_serial_pageRank(g, sol2, PageRankDampening, PageRankConvergence);
    ref_pagerank_time = CycleTimer::currentSeconds() - start;

    printf("----------------------------------------------------------\n");
    std::cout << "Testing Correctness of Page Rank\n";
    if (!compareApprox(g, sol2, sol1)) {
        pr_check = false;
    }

    if (!pr_check)
        std::cout << "Your Page Rank is not Correct" << std::endl;
    else
        std::cout << "Your Page Rank is Correct" << std::endl;

    char buf[1024];
    char ref_buf[1024];
    sprintf(buf, "%4d:   %.6f s\n", thread_count, pagerank_time);
    sprintf(ref_buf, "   1:   %.6f s\n", ref_pagerank_time);

    timing << buf;
    ref_timing << ref_buf;

    printf("----------------------------------------------------------\n");
    std::cout << "Serial Reference Summary" << std::endl;
    std::cout << ref_timing.str();

    printf("----------------------------------------------------------\n");
    std::cout << "Timing Summary" << std::endl;
    std::cout << timing.str();

    printf("----------------------------------------------------------\n");

    delete g;
    return 0;
}
```

# 3: OpenMP Programming

## Main.py

```
#include "your_reduce.h"
#include <cassert>
#include <cstdio>
#include <cstdlib>
#include <cstring>
#include <ctime>
#include <random>
#include <sys/time.h>

#define MAX_LEN 268435456

// return the time in the unit of us
static long get_time_us() {
    struct timeval my_time;
    gettimeofday(&my_time, NULL);
    long runtime_us = 1000000 * my_time.tv_sec + my_time.tv_usec;
    return runtime_us;
}

int main(int argc, char *argv[]) {
    int size, rank, provided;

    MPI_Init_thread(&argc, &argv, MPI_THREAD_MULTIPLE,
                    &provided); // enable multi-thread support (for Bonus)
    assert(provided == MPI_THREAD_MULTIPLE);

    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    int *a, *b; // the array used to do the reduction

    int *res;  // the array to record the result of YOUR_Reduce
    int *res2; // the array to record the result of MPI_Reduce

    long count;
    long begin_time, end_time, use_time,
        use_time2; // use_time for YOUR_Reduce & use_time2 for MPI_Reduce

    int i;

    // initialize
    a = (int *)malloc(MAX_LEN * sizeof(int));
    b = (int *)malloc(MAX_LEN * sizeof(int));
    res = (int *)malloc(MAX_LEN * sizeof(int));
    res2 = (int *)malloc(MAX_LEN * sizeof(int));
    memset(a, 0, MAX_LEN * sizeof(int));
    memset(b, 0, MAX_LEN * sizeof(int));
    memset(res, 0, MAX_LEN * sizeof(int));
    memset(res2, 0, MAX_LEN * sizeof(int));

    std::mt19937 rng;
    rng.seed(time(NULL)); // seed to generate the array randomly

    for (count = 1; count <= MAX_LEN;
         count *= 16) // length of array : [ 1  16  256  4'096  65'536 1'048'576
                      // 16'777'216  268'435'456 ]
    // do not report results for length 1
    {
        // the element of array is generated randomly
        for (i = 0; i < count; i++) {
            b[i] = a[i] = rng() % MAX_LEN;
        }
```

```
        // MPI_Reduce and then print the usetime, the result will be put in
        // res2[]
        MPI_Barrier(MPI_COMM_WORLD);
        begin_time = get_time_us();
        MPI_Reduce(a, res2, count, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);
        MPI_Barrier(MPI_COMM_WORLD);
        end_time = get_time_us();
        use_time2 = end_time - begin_time;
        if (rank == 0)
            printf("%ld int use_time : %ld us [MPI_Reduce]\n", count,
                   use_time2),
                fflush(stdout);

        // YOUR_Reduce and then print the usetime, the result should be put in
        // res[]
        MPI_Barrier(MPI_COMM_WORLD);
        begin_time = get_time_us();

        YOUR_Reduce(b, res, count);

        MPI_Barrier(MPI_COMM_WORLD);
        end_time = get_time_us();
        use_time = end_time - begin_time;
        if (rank == 0)
            printf("%ld int use_time : %ld us [YOUR_Reduce]\n", count,
                   use_time),
                fflush(stdout);

        // check the result of MPI_Reduce and YOUR_Reduce
        if (rank == 0) {
            int correctness = 1;
            for (i = 0; i < count; i++) {
                if (res2[i] != res[i]) {
                    correctness = 0;
                }
            }
            if (correctness == 0)
                printf("WRONG !!!\n"), fflush(stdout);
            else
                printf("CORRECT !\n"), fflush(stdout);
        }
    }

    MPI_Finalize();

    return 0;
}
```

## My Reduce function

```
#include <mpi.h>
#include <cstring>
#include <stdio.h>

void YOUR_Reduce(const int *sendbuf, int *recvbuf, int count) {
    int rank, size;

    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    // Initialize recvbuf with the values from sendbuf
    memcpy(recvbuf, sendbuf, count * sizeof(int));

    // Allocate temp_buffer once
    int* temp_buffer = new int[count];

    // Binary tree reduction
    for (int step = 1; step < size; step *= 2) {
        if (rank % (2 * step) == 0) {
            // Root process collects data
            if (rank + step < size) {
                MPI_Recv(temp_buffer, count, MPI_INT, rank + step, 0,
MPI_COMM_WORLD, MPI_STATUS_IGNORE);

                // Combine results using a loop
                for (int i = 0; i < count; i++) {
                    recvbuf[i] += temp_buffer[i];
                }
            }
        } else if (rank % step == 0) {
            // Send to parent
            MPI_Send(recvbuf, count, MPI_INT, rank - step, 0, MPI_COMM_WORLD);
            break;
        }
    }

    // Clean up
    delete[] temp_buffer;
}
```

## My Reduce-Sequential function

```
#include <mpi.h>
#include <cstring>
#include <stdio.h>
#include <omp.h> // Include OpenMP header

void YOUR_Reduce(const int *sendbuf, int *recvbuf, int count) {
    int rank, size;

    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    // Initialize recvbuf with the values from sendbuf
    memcpy(recvbuf, sendbuf, count * sizeof(int));

    // Allocate temp_buffer once
    int* temp_buffer = new int[count];

    // Binary tree reduction
    for (int step = 1; step < size; step *= 2) {
        if (rank % (2 * step) == 0) {
            // Root process collects data
            if (rank + step < size) {
                MPI_Request request;

                // Start non-blocking receive
                MPI_Irecv(temp_buffer, count, MPI_INT, rank + step, 0,
MPI_COMM_WORLD, &request);

                // Wait for the receive to complete
                MPI_Wait(&request, MPI_STATUS_IGNORE);

                // Combine results in parallel
                #pragma omp parallel for
                for (int i = 0; i < count; i++) {
                    recvbuf[i] += temp_buffer[i];
                }
            }
        } else if (rank % step == 0) {
            // Send to parent using non-blocking send
            MPI_Request request;
            MPI_Isend(recvbuf, count, MPI_INT, rank - step, 0, MPI_COMM_WORLD,
&request);
            break;
        }
    }

    // Clean up
    delete[] temp_buffer;
}
```

# 4: Google File System

## *1: GFS Questions*

### *1.1: How does the master node get the locations of each chunks at startup?*

At startup, the master node does not store the chunk location information persistently. Instead, it retrieves the location of each chunk by polling all chunkservers. Each chunkserver reports the chunks it holds, and the master node updates its information accordingly. Additionally, whenever a chunkserver joins the cluster, the master node updates the chunk locations. The master keeps this information updated by sending periodic HeartBeat messages, making sure it is updated on chunkplacement and the chunkserver status. This approach ensures that the master always has up-to-date information about chunk locations in the system.

### *1.2: What is the benefit of this approach comparing with the approach that the master persists this information?*

The main benefit of this approach is the reduction in complexity related to maintaining consistency between the master and the chunkservers. Persisting chunk location information would require the system to handle various events such as chunkserver failures, renaming, and rejoining, which can lead to stale or inconsistent information. By polling the chunkservers at startup and using regular HeartBeat messages, the master node avoids the issues of synchronization and ensures accurate, up-to-date chunk location information at all times. This also simplifies the system's design, making it more robust against common failures in a distributed environment.

## 2: Cluster calculations

We assume a cluster of 1000 servers, each server having 10 disks with 10TB storage capacity and 100MB/s I/O bandwidth per disk. The servers are connected by a 1Gbps (125MBps) ethernet cables, as nothing else is written I assume this bandwidth is per machine and not the total transfer cap in the system.

### *2.1: What is the minimum time required to recovery a node failure*

The total I/O bandwidth of all disks in a node is:

*Total bandwidth = 10disks ×100MB/s = 1000MB/s*

The network bandwidth available per node is:

*Network bandwidth = 1Gbps = 1024Mbps = 128MB/s*

Since the network bandwidth is lower than the total I/O bandwidth of the disks, the recovery time will be limited by the network speed. Furthermore, we assume that since we are calculating the minimum time requred, that all other 999 nodes will use their resources to assist upon a node failure. Assuming that the 999 remaining servers work in perfect parallel to recover the data from the failed node, the total network bandwidth available is

*Total bandwidth = 999 ×128MB/s = 127872MB/s*

To transfer 100 TB of data at this rate, the time required is:

$$\text{Time to recover } = \frac{100 \times 1024 \times 1024 MB}{127872 MBs} = 820s = 13{,}667m$$

Thus, the minimum time required to recover a node failure, assuming all other servers participate in the recovery, is approximately 14 minutes.

### *2.2: Time to recover a failure node with throttled recovery bandwidth*

Since the recovery traffic is throttled to 100 Mbps per machine, roughly a tenth of the original 1Gbps, we would expect a recovery time ten times as long, as the recovery time is directly proportional to the network. Here we still assume that all other 999 nodes, as nothing else is stated in the assignment. We also assume all other requirements are similar. Below are my calculations:

$$\text{Total bandwidth} = 999 \times \frac{100\,\text{Mbps}}{8\,\text{MB/Mb}} = 999 \times 12.5\,\text{MB/s} = 12487.5\,\text{MB/s}$$

At this throttled rate, the time required to recover 100 TB of data is:

$$\text{Time to recover} = \frac{100 \times 1024 \times 1024\,\text{MB}}{12487.5\,\text{MB/s}} = 8397\,\text{seconds}$$

Converting this to hours:

$$\text{Time to recover} = \frac{8392\,\text{seconds}}{3600\,\text{seconds/h}} \approx 2.33\,\text{hours.}$$

Thus, the time required to recover a node failure when the recovery traffic is throttled is approximately 2 hours and 20 minutes.

## *Q3: How many server failures are likely to occur in a year in this cluster? What is the mean time between node failures in this cluster?*

We know that each server node has a mean time between failures (MTBF) of 10,000 hours.

$$\frac{24\,\text{h/day} \times 365\,\text{days/year}}{10,000\,\text{h MTBF}} = \frac{8,760\,\text{h/year}}{10,000\,\text{h/failure}} = 0.876\,\text{failures/year/server}.$$

As this failure rate is independent for each server node, we have to multiply this by the total amount of serves in the cluster to estimate the expected number of failures per year in the cluster:

$$1000\,\text{servers} \times 0.876\,\text{failures/year/server} = 876\,\text{failures/year}.$$

The mean time between failures (MTBF) for the entire cluster is therefore:

$$\frac{8,760\,\text{hours/year}}{876\,\text{failures/year}} = 10\,\text{hours}.$$

Thus, the cluster is expected to experience one server failure approximately every 10 hours.

## *Q4: What is the implication of the number of replicas used in GFS based on the results from Q2 and Q3?*

The comparison between the recovery time (Q2) and the mean time between node failures (Q3) highlights the critical importance of replication in GFS. With recovery taking approximately 2.33 hours when throttled (as calculated in Q2), and with the cluster experiencing a node failure approximately every 10 hours (from Q3), it's evident that replication is essential to prevent data loss. The default number of replicas in GFS is three, which ensures redundancy and availability of data even when nodes fail.

The chances of multiple replicas experiencing data loss due to node failues are slim, and the chances of this is greatly reduces for each copy. Still, increasing the number leads to a reduction in performance as more updates have to be completed for each chunk write. Still, the default of three replicas strikes a good balance for most use cases. It provides sufficient fault tolerance while keeping the storage overhead manageable. Increasing the number of replicas could provide additional safety in environments with higher failure rates, but this would come at the cost of additional storage requirements. Conversely, reducing the number of replicas would expose the system to an increased risk of data loss in the event of multiple simultaneous failures, which GFS is designed to avoid.

# 5: MapReduce

## OutDegree.java

```
import java.io.IOException;
import java.util.StringTokenizer;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.Mapper;
import org.apache.hadoop.mapreduce.Reducer;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
import org.apache.hadoop.util.GenericOptionsParser;

public class OutDegree {

    public static class OutDegreeMapper
        extends Mapper<Object, Text, Text, IntWritable> {

        private final static IntWritable one = new IntWritable(1);
        private Text node = new Text();

        public void map(Object key, Text value, Context context
                        ) throws IOException, InterruptedException {
            StringTokenizer itr = new StringTokenizer(value.toString());
                                  itr.nextToken();       // Skip the first token ("a")
            if (itr.hasMoreTokens()) {
                String sourceNode = itr.nextToken(); // The source node (e.g., "a")
                node.set(sourceNode);
                // Emit the source node with a count of 1 for each outgoing edge
                context.write(node, one);
            }
        }
    }

    public static class OutDegreeReducer
        extends Reducer<Text, IntWritable, Text, IntWritable> {
        private IntWritable result = new IntWritable();

        public void reduce(Text key, Iterable<IntWritable> values,
                            Context context
                        ) throws IOException, InterruptedException {
            int sum = 0;
            for (IntWritable val : values) {
                sum += val.get(); // Sum up all counts for each node
            }
            result.set(sum);
            context.write(key, result); // Emit the node and its total out-degree
        }
    }

    public static void main(String[] args) throws Exception {
        Configuration conf = new Configuration();
        String[] otherArgs = new GenericOptionsParser(conf, args).getRemainingArgs();
        if (otherArgs.length < 2) {
            System.err.println("Usage: outdegree <in> <out>");
            System.exit(2);
        }
        Job job = new Job(conf, "outdegree");
        job.setJarByClass(OutDegree.class);
        job.setMapperClass(OutDegreeMapper.class);
        job.setCombinerClass(OutDegreeReducer.class);
        job.setReducerClass(OutDegreeReducer.class);
        job.setNumReduceTasks(1);
        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(IntWritable.class);
        for (int i = 0; i < otherArgs.length - 1; ++i) {
            FileInputFormat.addInputPath(job, new Path(otherArgs[i]));
        }
        FileOutputFormat.setOutputPath(job, new Path(otherArgs[otherArgs.length - 1]));
        System.exit(job.waitForCompletion(true) ? 0 : 1); }}
```

## WordCount.java

```java
import java.io.IOException;
import java.util.StringTokenizer;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.Mapper;
import org.apache.hadoop.mapreduce.Reducer;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;

public class WordCount {

    public static class TokenizerMapper extends Mapper<Object, Text, Text, IntWritable> {

        private final static IntWritable one = new IntWritable(1);
        private Text word = new Text();

        public void map(Object key, Text value, Context context) throws IOException,
InterruptedException {
            StringTokenizer itr = new StringTokenizer(value.toString());
            while (itr.hasMoreTokens()) {
                word.set(itr.nextToken());
                context.write(word, one);
            }
        }
    }

    public static class IntSumReducer extends Reducer<Text, IntWritable, Text, IntWritable> {
        private IntWritable result = new IntWritable();

        public void reduce(Text key, Iterable<IntWritable> values, Context context)
                throws IOException, InterruptedException {
            int sum = 0;
            for (IntWritable val : values) {
                sum += val.get();
            }
            result.set(sum);
            context.write(key, result);
        }
    }

    public static int run(String input, String output) throws Exception {
        Configuration conf = new Configuration();
        Job job = Job.getInstance(conf, "word count");
        job.setJarByClass(WordCount.class);
        job.setMapperClass(TokenizerMapper.class);
        job.setCombinerClass(IntSumReducer.class);
        job.setReducerClass(IntSumReducer.class);
        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(IntWritable.class);
        FileInputFormat.addInputPath(job, new Path(input));
        FileOutputFormat.setOutputPath(job, new Path(output));
        return job.waitForCompletion(true) ? 0 : -1;
    }

    public static void main(String[] args) throws Exception {
        if (args.length != 2) {
            System.err.println("Usage: WordCount <in> <frontiers> <out>");
            System.exit(2);
        }
        if (run(args[0], args[1]) < 0) {
            System.exit(1);
        }
    }
}
```

# 6: Spark Programming

## PageRank:

```
import sys
from pyspark import SparkConf, SparkContext
import time

if __name__ == '__main__':
    conf = SparkConf()
    sc = SparkContext(conf=conf)
    sc.setLogLevel("ERROR") # Added to avoid Warnings cluttering the terminal
    file_path = sys.argv[1]
    lines = sc.textFile(file_path)

    # Parameters
    damping = 0.8
    num_iterations = 50
    top_i_nodes = 5
    dynamic = sys.argv[2]

    first = time.time()

    # Parse the lines into (source, destination) pairs and remove duplicates
    edges = lines.map(lambda line: tuple(map(int, line.split()))).distinct()

    # We estimate the amount of nodes
    if(dynamic):
        # This method takes aprox. 0.5s longer but is dynamic
        nodes = edges.flatMap(lambda edge: edge).distinct()
        n = nodes.count()
    else:
        # This method is faster but not dynamic
        n = 100 if "small" in file_path else 1000 if "full" in file_path else None

    # Create an adjacency list as (node, [neighbors])
    adj_list = edges.groupByKey().mapValues(list).cache()

    # Initialize each node's PageRank value
    page_ranks = adj_list.mapValues(lambda _: 1.0 / n)

    for i in range(num_iterations):
        # Broadcast the adjacency list for efficient access
        adjacency_broadcast = sc.broadcast(adj_list.collectAsMap())

        # Compute contributions for each node's neighbors
        contributions = page_ranks.flatMap(lambda node_rank: [
            (neighbor, node_rank[1] /
len(adjacency_broadcast.value.get(node_rank[0], [])))
            for neighbor in adjacency_broadcast.value.get(node_rank[0], [])
        ])

        # Aggregate contributions and calculate new PageRank values
        page_ranks = contributions.reduceByKey(lambda a, b: a + b).mapValues(
            lambda rank: (1 - damping) / n + damping * rank
        )

    # Get the top 5 nodes with the highest PageRank scores
    highest = page_ranks.takeOrdered(top_i_nodes, key=lambda x: -x[1])

    # Print the top 5 nodes
    print("Top 5 nodes with highest PageRank scores:")
    for node, score in highest:
        print(f"Node {node}: {score}")

    last = time.time()
    print("Total program time: %.2f seconds" % (last - first))
```

```
    sc.stop()
```

## WordCount:

```
import re
import sys
from pyspark import SparkConf, SparkContext
import time

if __name__ == '__main__':
    conf = SparkConf()
    sc = SparkContext(conf=conf)
    sc.setLogLevel("ERROR") # Added to avoid Warnings cluttering the terminal
    lines = sc.textFile(sys.argv[1])
    top_i_words = 10

    first = time.time()

    # We split the lines into words
    words = lines.flatMap(lambda line: re.split(r'[^\w]+', line))

    # We now count every word
    word_counts = words.countByValue()
    top_words = sorted(word_counts.items(), key=lambda x: -x[1])[:top_i_words]

    # Print the top i words
    print(f"Top {top_i_words} words:")
    for word, count in top_words:
        print(f"({repr(word)}, {count})")

    last = time.time()

    print("Total program time: %.2f seconds" % (last - first))
    sc.stop()
```

# 7: Worse is better

## What I learned

I find the article “Worse is Better” by Richard P. Gabriel interesting as, while somewhat dated, it gives an interesting view into the compromises and decisions made in the early days of programming and computer science.

The article explores two contrasting design philosophies in software development: the MIT approach, which focuses on achieving a complete and ideal “right” design, and the “worse-is-better” philosophy, also called the New Jersey approach, which values simplicity and practicality. Gabriel argues that simpler, less complete designs (termed ”worse”) are often more successful in real-world applications, as they foster adaptability and ease of use. This approach prioritizes implementation simplicity over full correctness or completeness, leading to systems that are easier to port and modify. He illustrates this with Unix and C, where simplicity and minimal resource requirements enable these systems to thrive across various environments. The title, **The Rise of Worse is Better**, reflects how “worse” design choices—those sacrificing some ideal goals—often yield “better” real-world success through increased portability, adaptability, and longevity. In this context, “worse” signifies compromises in design ideals, while “better” represents the widespread adoption and durability achieved by these practical systems.

# 8: Graph Partitioning

## GraphPartitioner.py:

```
import sys
import struct
import random
from collections import defaultdict
import matplotlib.pyplot as plt
import networkx as nx

class GraphPartitioner:
    def __init__(self, file_path, num_partitions, show_details):
        self.file_path = file_path
        self.num_partitions = num_partitions
        self.show_details = show_details
        self.edges = self.read_graph()
        self.directed = self.isDirected()

    def isDirected(self):
        # Some of the graphs are undirected
        return not any(file_name in file_path for file_name in ["small-5",
"synthesized-1b"])

    # A (Slightly modified) replication of the C code already existing
    # Reads the data and saves all edges in the graph
    def read_graph(self):
        edges = []
        with open(self.file_path, "rb") as file:
            data = file.read(8)     # Read first 8 bytes
            while data:
                src, dst = struct.unpack("ii", data) # (4 byte src, 4 byte dst)
                edges.append((src, dst))
                data = file.read(8) # Read the next 8 bytes
        print("Done reading edges\n")
        return edges

    # Show graphs in subplots for each partition
    def show_graph(self, partitions, title="Graph Partitions"):
        height = 4
        width = len(partitions) * height
        fig, axes = plt.subplots(1, len(partitions), figsize=(width, height))

        if len(partitions) == 1:
            axes = [axes]  # Ensure axes is always iterable for a single partition

        for i, (partition_id, partition) in enumerate(sorted(partitions.items())):
            ax = axes[i]
            G = nx.DiGraph() if self.directed else nx.Graph()
            G.add_edges_from(partition['edges'])

            # Differentiate master vertices by color, or color all nodes lightgreen
if none exist
            master_vertices = partition.get('master_vertices', set())
            # If there is no master_vertices value, then we are on the Full graph
and all are master vertices
            if not master_vertices:
                node_colors = ["lightgreen" for _ in G.nodes()]
            # If only some are master vertices, color them lightgreen and the rest
skyblue
            else:
                node_colors = ["lightgreen" if node in master_vertices else
"skyblue" for node in G.nodes()]

            pos = nx.spring_layout(G)  # Use spring layout for better visualization
            nx.draw(G, pos, with_labels=True, node_color=node_colors,
font_weight="bold",
```

```
                node_size=500, edge_color="gray", ax=ax, arrows=self.directed)
            ax.set_title(f"Partition {partition_id}")

        fig.suptitle(title, fontsize=16)
        plt.tight_layout(rect=[0, 0, 1, 0.95])
        plt.show()

    # Balanced p-way edge-cut partitioning
    def edge_cut_partition(self):
        partitions = defaultdict(
            lambda: {
                'master_vertices': set(),
                'total_vertices': set(),
                'replicated_edges': 0,
                'edges': []
            }
        )
        vertex_to_partition = {}

        for src, dst in self.edges:
            # Hash vertices to assign them to partitions
            src_partition = vertex_to_partition.get(src, hash(src) %
self.num_partitions)
            dst_partition = vertex_to_partition.get(dst, hash(dst) %
self.num_partitions)

            # Assign the vertex to the partition
            vertex_to_partition[src] = src_partition
            vertex_to_partition[dst] = dst_partition

            # Assign edges and count replicated edges
            if src_partition != dst_partition:
                partitions[src_partition]['replicated_edges'] += 1
                partitions[dst_partition]['replicated_edges'] += 1

            partitions[src_partition]['edges'].append((src, dst))
            partitions[dst_partition]['edges'].append((src, dst))

            # Update master and total vertices
            partitions[src_partition]['master_vertices'].add(src)
            partitions[dst_partition]['master_vertices'].add(dst)
            partitions[src_partition]['total_vertices'].update([src, dst])
            partitions[dst_partition]['total_vertices'].update([src, dst])

        # Print partition statistics
        for i, partition in sorted(partitions.items()):
            print(f"Partition {i}")
            print(len(partition['master_vertices']))
            print(len(partition['total_vertices']))
            print(partition['replicated_edges'])
            print(len(partition['edges']))

        # Show the partitions side by side
        if self.show_details:
            self.show_graph(partitions, title="Edge-Cut Partitioning")

    # Balanced p-way vertex-cut partitioning
    def vertex_cut_partition(self):
        partitions = defaultdict(
            lambda: {
                'master_vertices': set(),
                'total_vertices': set(),
                'edges': []
            }
        )
        vertex_partitions = defaultdict(set)  # Store all partitions a vertex is
assigned to
```

```
        for src, dst in self.edges:
            # Hash the edge to assign it to a partition
            edge_partition = hash((src, dst)) % self.num_partitions
            partitions[edge_partition]['edges'].append((src, dst))

            # Update vertex partitions
            vertex_partitions[src].add(edge_partition)
            vertex_partitions[dst].add(edge_partition)

        # Calculate master and total vertices for each partition
        for vertex, assigned_partitions in vertex_partitions.items():
            # Choose one partition as master
            master_partition = random.choice(list(assigned_partitions))
            for partition_id in assigned_partitions:
                partitions[partition_id]['total_vertices'].add(vertex)
                if partition_id == master_partition:
                    partitions[partition_id]['master_vertices'].add(vertex)

        # Print partition statistics
        for i, partition in sorted(partitions.items()):
            print(f"Partition {i}")
            print(f"{len(partition['master_vertices'])}")
            print(f"{len(partition['total_vertices'])}")
            print(f"{len(partition['edges'])}")

        # Show the partitions side by side
        if self.show_details:
            self.show_graph(partitions, title="Vertex-Cut Partitioning")

    # Greedy heuristic vertex-cut partitioning
    def greedy_heuristic_partition(self):
        partitions = defaultdict(
            lambda: {
                'master_vertices': set(),
                'total_vertices': set(),
                'edges': []
            }
        )
        vertex_degrees = defaultdict(int)

        # Calculate degrees for all vertices
        for src, dst in self.edges:
            vertex_degrees[src] += 1
            vertex_degrees[dst] += 1

        # Assign vertices to partitions using greedy heuristic
        for src, dst in self.edges:
            src_partition = vertex_degrees[src] % self.num_partitions
            dst_partition = vertex_degrees[dst] % self.num_partitions

            partitions[src_partition]['edges'].append((src, dst))
            partitions[dst_partition]['edges'].append((src, dst))

            partitions[src_partition]['total_vertices'].add(src)
            partitions[dst_partition]['total_vertices'].add(dst)

            master_partition = random.choice([src_partition, dst_partition])
            if master_partition == src_partition:
                partitions[src_partition]['master_vertices'].add(src)
            else:
                partitions[dst_partition]['master_vertices'].add(dst)

        # Ensure every partition gets at least one vertex
        for i in range(self.num_partitions):
            if not partitions[i]['total_vertices']:
                random_vertex = random.choice(list(vertex_degrees.keys()))
                partitions[i]['total_vertices'].add(random_vertex)
                partitions[i]['master_vertices'].add(random_vertex)
```

```
        # Print partition statistics
        for i, partition in sorted(partitions.items()):
            print(f"Partition {i}")
            print(len(partition['master_vertices']))
            print(len(partition['total_vertices']))
            print(len(partition['edges']))

        # Show the partitions side by side
        if self.show_details:
            self.show_graph(partitions, title="Greedy Vertex-Cut Partitioning")

    # Helper function to get the neighbors of a vertex
    def get_neighbors(self, vertex):
        neighbors = set()
        for src, dst in self.edges:
            if src == vertex:
                neighbors.add(dst)
            elif dst == vertex:
                neighbors.add(src)
        return neighbors

    # Helper function to get partition for a vertex
    def get_partition_for_vertex(self, vertex, partitions):
        for partition_id, partition in partitions.items():
            if vertex in partition['total_vertices']:
                return partition_id
        return None

    # Balanced p-way Hybrid-Cut based on PowerLyra
    def hybrid_cut_partition(self, theta=10):
        partitions = defaultdict(
            lambda: {
                'master_vertices': set(),
                'total_vertices': set(),
                'edges': []
            }
        )
        vertex_degree = defaultdict(int)  # Store vertex degrees
        vertex_partitions = defaultdict(set)  # Tracks the partitions each vertex
is assigned to

        # Calculate degrees for all vertices
        for src, dst in self.edges:
            vertex_degree[src] += 1
            vertex_degree[dst] += 1

        for src, dst in self.edges:
            if vertex_degree[src] > theta or vertex_degree[dst] > theta:
                # High-degree vertices: use edge-cut strategy
                src_partition = hash(src) % self.num_partitions
                dst_partition = hash(dst) % self.num_partitions
                partitions[src_partition]['edges'].append((src, dst))
                partitions[dst_partition]['edges'].append((src, dst))
                partitions[src_partition]['master_vertices'].add(src)
                partitions[dst_partition]['master_vertices'].add(dst)
                partitions[src_partition]['total_vertices'].update([src, dst])
                partitions[dst_partition]['total_vertices'].update([src, dst])
            else:
                # Low-degree vertices: use vertex-cut (greedy heuristic)
                min_partition = None
                min_replication = float('inf')

                for partition_id in range(self.num_partitions):
                    replication_cost = (
                        int(src not in partitions[partition_id]['master_vertices'])
+
                        int(dst not in partitions[partition_id]['master_vertices'])
```

```
                    )
                    if replication_cost < min_replication:
                        min_replication = replication_cost
                        min_partition = partition_id

                partitions[min_partition]['edges'].append((src, dst))
                partitions[min_partition]['master_vertices'].update([src, dst])
                partitions[min_partition]['total_vertices'].update([src, dst])
                vertex_partitions[src].add(min_partition)
                vertex_partitions[dst].add(min_partition)

        # Print partition statistics
        for i, partition in sorted(partitions.items()):
            print(f"Partition {i}")
            print(f"{len(partition['master_vertices'])}")
            print(f"{len(partition['total_vertices'])}")
            print(f"{len(partition['edges'])}")

        if self.show_details:
            self.show_graph(partitions, title=f"Hybrid-Cut Partitioning
(Theta={theta})")


if __name__ == "__main__":
    file_path = sys.argv[1] if len(sys.argv) > 1 else "small-5.graph"
    num_partitions = int(sys.argv[2]) if len(sys.argv) > 2 else 3
    show_details = (sys.argv[3]!="False") if len(sys.argv) > 3 else False
    partitioner = GraphPartitioner(file_path, num_partitions, show_details)

    # Display the graph if applicable
    if show_details:
        partitioner.show_graph({0: {'edges': partitioner.edges}}, title="Full
Graph")

    # Partition the graph
    print("\nEdge-Cut Partitioning:")
    partitioner.edge_cut_partition()
    print("\nVertex-Cut Partitioning:")
    partitioner.vertex_cut_partition()
    print("\nHybrid-Cut Partitioning:")
    partitioner.hybrid_cut_partition(theta=2)
    print("\nGreedy-Cut Partitioning:")
    partitioner.greedy_heuristic_partition()
```

# 9: Graph algorithms with GridGraph

## Kcores.cpp:

```
#include "core/graph.hpp"

int main(int argc, char **argv) {
    if (argc < 3) {
        fprintf(stderr, "usage: kcores [path] [k] [memory budget in GB]\n");
        exit(-1);
    }

    std::string path = argv[1];
    int k = atoi(argv[2]);
    long memory_bytes = (argc >= 4) ? atol(argv[3]) * 1024l * 1024l * 1024l : 8l *
1024l * 1024l * 1024l;

    Graph graph(path);
    graph.set_memory_bytes(memory_bytes);

    Bitmap *active_in = graph.alloc_bitmap();
    Bitmap *active_out = graph.alloc_bitmap();
    BigVector<int> degree(graph.path + "/degree", graph.vertices);
    BigVector<int> core(graph.path + "/core", graph.vertices);

    long vertex_data_bytes = (long)graph.vertices * (sizeof(int) + sizeof(int));
    graph.set_vertex_data_bytes(vertex_data_bytes);

    // Initialize degree and active vertices
    active_out->fill();
    degree.fill(0);
    graph.stream_edges<VertexId>(
        [&](Edge &e) {
            write_add(&degree[e.source], 1);
            return 0;
        },
        nullptr, 0, 0);

    // Initialize core and set initial active vertices
    int active_vertices = graph.stream_vertices<VertexId>(
        [&](VertexId i) {
            core[i] = (degree[i] >= k) ? 1 : 0;
            return core[i];
        });

    printf("Initialization complete: %d active vertices\n", active_vertices);

    // K-core decomposition iterations
    int iteration = 0;
    while (active_vertices > 0) {
        iteration++;
        printf("Iteration %d: %d active vertices\n", iteration, active_vertices);

        std::swap(active_in, active_out);
        active_out->clear();

        graph.hint(degree, core);
        active_vertices = graph.stream_edges<VertexId>(
            [&](Edge &e) {
                if (core[e.source] == 1 && core[e.target] == 0) {
                    write_add(&degree[e.target], -1);
                    if (degree[e.target] < k) {
                        core[e.target] = 0;
                        active_out->set_bit(e.target);
                        return 1;
                    }
                }
```

```
                return 0;
            },
            active_in);
    }

    // Count k-core vertices
    int kcore_vertices = graph.stream_vertices<VertexId>(
        [&](VertexId i) {
            return core[i] == 1;
        });

    printf("K-core (%d-core) decomposition complete: %d vertices remain\n", k,
kcore_vertices);

    return 0;
}
```

## PageRankDelta.cpp:

```
#include "core/graph.hpp"

int main(int argc, char ** argv) {
       if (argc < 3) {
              fprintf(stderr, "usage: pagerank_delta [path] [iterations] [memory
budget in GB]\n");
              exit(-1);
       }
       std::string path = argv[1];
       int iterations = atoi(argv[2]);
       long memory_bytes = (argc >= 4) ? atol(argv[3]) * 1024l * 1024l * 1024l : 8l
* 1024l * 1024l * 1024l;

       Graph graph(path);
       graph.set_memory_bytes(memory_bytes);

       BigVector<VertexId> degree(graph.path + "/degree", graph.vertices);
       BigVector<float> pagerank(graph.path + "/pagerank", graph.vertices);
       BigVector<float> delta(graph.path + "/delta", graph.vertices);
       BigVector<float> new_delta(graph.path + "/new_delta", graph.vertices);

       long vertex_data_bytes = (long)graph.vertices * (sizeof(VertexId) +
sizeof(float) * 3);
       graph.set_vertex_data_bytes(vertex_data_bytes);

       double begin_time = get_time();

       // Initialize degrees
       degree.fill(0);
       graph.stream_edges<VertexId>(
              [&](Edge & e) {
                     write_add(&degree[e.source], 1);
                     return 0;
              }, nullptr, 0, 0
       );

       // Initialize Pagerank and Delta
       graph.hint(pagerank, delta, new_delta);
       graph.stream_vertices<VertexId>(
              [&](VertexId i) {
                     pagerank[i] = 0.15f;  // Initial PageRank value
                     delta[i] = 1.0f / degree[i];  // Initial delta
                     new_delta[i] = 0;
                     return 0;
              }, nullptr, 0,
              [&](std::pair<VertexId, VertexId> vid_range) {
                     pagerank.load(vid_range.first, vid_range.second);
                     delta.load(vid_range.first, vid_range.second);
```

```
			new_delta.load(vid_range.first, vid_range.second);
		},
		[&](std::pair<VertexId, VertexId> vid_range) {
			pagerank.save();
			delta.save();
			new_delta.save();
		}
	);

	// PageRank Delta Iterations
	for (int iter = 0; iter < iterations; iter++) {
		graph.hint(delta, new_delta);
		graph.stream_edges<VertexId>(
			[&](Edge & e) {
				write_add(&new_delta[e.target], 0.85f * delta[e.source]);
				return 0;
			}, nullptr, 0, 1,
			[&](std::pair<VertexId, VertexId> source_vid_range) {
				delta.lock(source_vid_range.first,
source_vid_range.second);
			},
			[&](std::pair<VertexId, VertexId> source_vid_range) {
				delta.unlock(source_vid_range.first,
source_vid_range.second);
			}
		);

		graph.hint(pagerank, delta, new_delta);
		graph.stream_vertices<float>(
			[&](VertexId i) {
				pagerank[i] += new_delta[i];
				delta[i] = new_delta[i] / degree[i];
				new_delta[i] = 0;  // Reset for next iteration
				return 0;
			}, nullptr, 0,
			[&](std::pair<VertexId, VertexId> vid_range) {
				pagerank.load(vid_range.first, vid_range.second);
				delta.load(vid_range.first, vid_range.second);
				new_delta.load(vid_range.first, vid_range.second);
			},
			[&](std::pair<VertexId, VertexId> vid_range) {
				pagerank.save();
				delta.save();
				new_delta.save();
			}
		);
	}

	double end_time = get_time();
	printf("%d iterations of pagerank delta took %.2f seconds\n", iterations,
end_time - begin_time);

	return 0;
}
```

# 10: Graph Mining

## FSM.py

```
import pandas as pd
import numpy as np
import json
from collections import defaultdict
from itertools import combinations
from networkx.algorithms import isomorphism
from math import comb
from multiprocessing import Pool

def build_graph(edges, vertices):
    print("Building graph...")
    graph = defaultdict(list)

    # Ensure unique indices in the vertices DataFrame
    vertices = vertices.drop_duplicates(subset='id').set_index('id')

    # Add all edges to the graph
    for _, row in edges.iterrows():
        source, target = row['source_id'], row['target_id']
        graph[source].append((target, row['amt'], row['strategy_name'],
row['buscode']))

    print(f"Total nodes: {len(graph)}, Total edges: {sum(len(v) for v in
graph.values())}\n")
    return graph

def hash_edge(source, target, amt, strategy_name, buscode):
    # I think we can go without some of these values, but thought they might be
nice to have anyways :-)
    return f"{min(source, target)}-{max(source, target)}-{amt}-{strategy_name}-
{buscode}"

def mine_frequent_subgraphs(graph, pattern_size, support_threshold, output_file):
    print("Mining frequent subgraphs...")
    subgraph_counts = defaultdict(int)
    subgraph_patterns = defaultdict(list)

    # Iterate over node combinations
    for nodes in combinations(graph.keys(), pattern_size):
        edges = []
        for u, v in combinations(nodes, 2):
            if v in [neighbor[0] for neighbor in graph[u]]:
                edge_details = next(
                    (neighbor for neighbor in graph[u] if neighbor[0] == v), None
                )
                if edge_details:
                    edges.append(
                        hash_edge(u, v, *edge_details[1:])
                    )

        if len(edges) == pattern_size:
            subgraph_key = "_".join(sorted(edges))
            subgraph_counts[subgraph_key] += 1
            subgraph_patterns[subgraph_key].append(edges)

    # Filter frequent subgraphs
    frequent_subgraphs = {
        k: v for k, v in subgraph_counts.items() if v >= support_threshold
    }

    save_results(frequent_subgraphs, subgraph_patterns, output_file)
    print("Frequent subgraph mining completed.")
```

```
def save_results(frequent_subgraphs, subgraph_patterns, output_file):
    result = []
    for subgraph, frequency in frequent_subgraphs.items():
        edges = subgraph_patterns[subgraph]
        result.append({
            "frequency": frequency,
            "edges": [{"source": edge.split("-")[0], "target": edge.split("-")[1],
"details": edge} for edge in edges]
        })

    with open(output_file, 'w') as f:
        json.dump(result, f, indent=4)
    print(f"Results saved to {output_file}")

if __name__ == "__main__":
    print("Reading data...")
    header1 = ['id', 'name', 'timestamp', 'black']
    header2 = ['source_id', 'target_id', 'timestamp', 'amt', 'strategy_name',
'trade_no', 'buscode', 'other']

    account = pd.read_csv('data/account', names=header1, sep=',')
    card = pd.read_csv('data/card', names=header1, sep=',')
    account_to_account = pd.read_csv('data/account_to_account', names=header2,
sep=',', usecols=range(len(header2)))
    account_to_card = pd.read_csv('data/account_to_card', names=header2, sep=',',
usecols=range(len(header2)))

    vertices = pd.concat([account, card])
    edges = pd.concat([account_to_account, account_to_card])
    edges['amt'] = edges['amt'].round()

    graph = build_graph(edges, vertices)

    # Start mining frequent subgraphs
    mine_frequent_subgraphs(
        graph,
        pattern_size=3,
        support_threshold=10000,
        output_file=f"results/bdci_data.json"
    )
```

# 11: Spark Streaming Top-K

## Top-k.py:

```
from __future__ import print_function
import sys
from pyspark import SparkContext
from pyspark.streaming import StreamingContext
from pyspark.streaming.dstream import DStream

# --- Code Setup ---

# Initialize SparkContext and StreamingContext
sc = SparkContext(appName="Py_HDFSWordCount")
ssc = StreamingContext(sc, 60)

# Create a DStream that listens to the HDFS directory
hdfs_directory = "hdfs://intro00:8020/user/2024403421/stream"
lines = ssc.textFileStream(hdfs_directory)

# We use these global variables to keep track of our top-k algorithm
word_counts = {}
file_no = 1
last_file = 5
k = 100

# --- My two functions to perform the Top-K ---

# Function to update the word counts with each new RDD
def update_count(new_counts, last_counts):
    # On first batch we initialize an empty dictionary
    if last_counts is None:
        last_counts = {}

    # On all continuing files, update the word counts
    for word, count in new_counts:
        if word in last_counts:
            last_counts[word] += count
        else:
            last_counts[word] = count

    return last_counts

# Function to process each RDD and compute the top-k frequent words
def process_rdd(time, rdd):
    global word_counts, file_no, last_file, k
    if rdd.isEmpty():
        return

    # Compute word counts for the current RDD
    counts = rdd.flatMap(lambda line: line.split(" ")) \
                .map(lambda word: (word, 1)) \
                .reduceByKey(lambda a, b: a + b)

    # Update the global word counts using the updateCounts function
    updated_counts = counts.collect()  # Collect current counts to update
    word_counts = update_count(updated_counts, word_counts)

    # Sort by count and take top k
    top_k = sorted(word_counts.items(), key=lambda x: x[1], reverse=True)[:k]

    # We structure the output
    output = f"----- File Number {file_no} -----\n"
    output += f" --- Top-{k} words so far ---\n"

    for word, count in top_k:
        output += f"{word}: {count}\n"
```

```
    # We write the output to a file
    output_filename = f"output_file_{file_no}.txt"
    with open(output_filename, "w") as f:
        f.write(output)

    print(output)

    file_no += 1

    # Stop the program after processing the last file
    if file_no > last_file:
        print("\nMaximum number of files processed. Stopping the streaming.")
        ssc.stop(stopSparkContext=True, stopGraceFully=True)

# --- Use My functions ---

# Process each RDD in the DStream and compute top-k
lines.foreachRDD(process_rdd)

# Start streaming and wait for termination
ssc.start()
ssc.awaitTermination()
```

# 12: Halide Scheduling

## Dialated-conv.cpp:

```
#include "Halide.h"
#include "common.h"

#include <stdio.h>

using namespace Halide;
using namespace Halide::Tools;

int main(int argc, char **argv) {
    const int N = 5, CI = 128, CO = 128, W = 100, H = 80, KW = 3, KH = 3;
    const int dilation = 15;

    ImageParam input(type_of<float>(), 4);
    ImageParam filter(type_of<float>(), 4);

    // Define variables and reduction domain
    Var x("x"), y("y"), c("c"), n("n");
    Var xo("xo"), yo("yo"), xi("xi"), yi("yi");
    Var co("co"), ci("ci");

    Func dilated_conv("dilated_conv");
    RDom r(0, CI, 0, KW, 0, KH);

    // Algorithm definition
    dilated_conv(c, x, y, n) = 0.0f;
    dilated_conv(c, x, y, n) += filter(c, r.y, r.z, r.x) *
        input(r.x, x + r.y * (dilation + 1), y + r.z * (dilation + 1), n);

    // **Scheduling**
    // 1. Split the x and y dimensions for tiling
    dilated_conv.compute_root()
        .tile(x, y, xo, yo, xi, yi, 8, 8)  // 8x8 tile size (tunable)
        .fuse(xo, yo, co)
        .parallel(co)                     // Parallelize outer loop over tiles
        .vectorize(xi, 8);                // Vectorize inner x dimension

    // 2. Optimize the reduction
    dilated_conv.update()
        .reorder(r.x, r.y, r.z, c, xi, yi, n)
        .unroll(r.z)                      // Unroll kernel height loop
        .unroll(r.y)                      // Unroll kernel width loop
        .vectorize(xi, 8)                 // Vectorize inner computation
        .parallel(n);                     // Parallelize across batch dimension

    // Buffer initialization
    Buffer<float, 4> in(CI, W + (KW - 1) * (dilation + 1), H + (KH - 1) * (dilation
+ 1), N);
    Buffer<float, 4> fil(CO, KW, KH, CI);
    Buffer<float, 4> output_halide(CO, W, H, N);

    // Initialize input and filter with random data
    random_data<float, 4>(in);
    random_data<float, 4>(fil);
    input.set(in);
    filter.set(fil);

    // JIT compile and run
    dilated_conv.realize(output_halide);
    double t_halide = benchmark(10, 10, [&]()
{ dilated_conv.realize(output_halide); });

    Buffer<float, 4> output_ref(CO, W, H, N);
```

```
    double t_onednn = dnnl_dilated_conv_wrapper(in.data(), fil.data(),
output_ref.data(),
                                                {N, CI, CO, W, H, KW, KH, dilation,
dilation});

    // Check correctness
    if (check_equal<float, 4>(output_ref, output_halide)) {
        printf("Halide results - OK\n");
    } else {
        printf("Halide results - FAIL\n");
        return 1;
    }

    float gflops = 2.0f * (N * CO * H * W) * (CI * KH * KW) / 1e9f;
    printf("Halide: %fms, %f GFLOP/s\n", t_halide * 1e3, (gflops / t_halide));
    printf("oneDNN: %fms, %f GFLOP/s\n\n", t_onednn * 1e3, (gflops / t_onednn));

    printf("Success!\n");
    return 0;
}
```